ILPs for Routing and Wavelength Assignment	01/2010

Part 1: Introduction to Linear Programming and GNU MathProg

Neal Charbonneau

Outline

This document goes over the basics of linear programming. The main focus is on formulating problems as linear programs (LPs) and writing MathProg scripts that can be used to solve the LPs. MathProg is a modeling language that is accepted by different LP solvers as input. In Part 2, we will focus more on networking applications and using CPLEX.

Introduction to Linear Programming

Linear Programming (LP) is a mathematical technique to find optimal solutions to a given problem. An LP allows you to minimize or maximize some objective (e.g. profits) given limited resources and a set of constraints. LPs are an *optimization* technique used to solve many problems in science, engineering, economics, etc. The “programming” in Linear Programming has nothing to do with computer programming, but refers to mathematical programming, or optimization.

There are many free and commercial LP solvers available. This means that someone can formulate their problem as an LP then use a solver to solve it instead of writing a custom application to solve their problem. LP solvers are very efficient so this can save time and money.

Example LP

We’ll start with an example problem that can be solved using a linear program. The example problem is from CLRS [1]. Consider a politician running in a mayoral race. The city can be divided into three types of areas—urban, suburban, and rural. Each of these areas has 100,000, 200,000 and 50,000 registered voters, respectively. Assume that in order to win the election, the candidate must win a majority in each of the three regions. The candidate is running on four issues, which are building new roads, gun control, farm subsidies, and a gasoline tax. Each of these four issues will have a different impact on different areas. For example, farm subsidies will be very popular for rural voters. The candidate must spend a certain amount of money on advertising for each of the issues. The campaign research staff has determined how many voters can be won or lost by spending $1,000 dollars of advertising showing support for each of the four issues. The information is shown in Table I. Negative values represent votes lost.

The problem now is to determine the minimum amount of money to spend in order to win a majority of votes from each area, that is 50,000 urban votes, 100,000 suburban votes, and 25,000 rural votes.

Table 1: Number of votes gained per $1,000 spent on advertising.

Policy	Urban	Suburban	Rural
Building Roads	-2,000	5,000	3,000
Gun Control	8,000	2,000	-5,000
Farm Subsidies	0	0	10,000
Gasoline Tax	10,000	0	-2,000

One way to do this is by trial and error. For example, say we decide to spend $20,000 on advertising for building roads, $0 for gun control, $4,000 for farm subsidies, and $9,000 for the gasoline tax. Using the columns of Table I, this will result in $20(-2000)+0(8000)+4(0)+9(10000) = 50,000$ votes for the urban population, $20(5000)+0(2000)+4(0)+9(0) = 100,000$ votes for suburban population, and lastly $20(3000) + 0(-5000) + 4(10000) + 9(-2000) = 82,000$ votes for the rural population. With this solution, we spent $20+0+4+9 = \$33,000$ on advertising and will get all of the votes needed to win the election.

Is it possible to spend even less money and still get the required votes? Even if we are able to find a lower cost solution, is it the minimum? It is not likely we can determine this using trial and error. To help us answer this question, we can formulate the problem as a linear program and then use an LP solver to find the optimal solution for us.

We want to determine the amount of money to spend on each of the four issues. We will represent each amount by a variable. Let x_1 be the amount of money spent on advertising the building of roads. Let x_2 be the amount spend on advertising gun control. x_3 and x_4 are the amount spent on advertising farm subsidies and the gasoline tax, respectively. The goal of the problem is to minimize the amount of money spent. In other words, we'd like to minimize $x_1 + x_2 + x_3 + x_4$. We need to minimize this sum, but we need to do so while gaining the amount of votes required. These can be formulated as constraints. For example, consider the case of gaining 50,000 urban voters. This can be formulated as

$$-2,000x_1 + 8,000x_2 + 0x_3 + 10,000x_4 \geq 50,000 \quad (1)$$

According to Table 1, for every thousand dollars we spend on advertising building roads (x_1), we lose 2,000 urban voters, which explains the $-2,000x_1$. Similarly we gain 8,000 votes for every thousand spent on gun control, and so on. The end result is that we need at least 50,000 urban votes. We can create similar equations to ensure we gain enough votes for suburban votes

$$5,000x_1 + 2,000x_2 + 0x_3 + 0x_4 \geq 100,000 \quad (2)$$

and rural votes

$$3,000x_1 + -5,000x_2 + 10,000x_3 + -2,000x_4 \geq 25,000. \quad (3)$$

Lastly, we cannot spend a negative amount of money advertising on any of the issues, so we must have

$$x_1, x_2, x_3, x_4 \geq 0. \tag{4}$$

We can combine equations (1-4) to create our linear program. The LP is shown below. It consists of a linear function that must be minimized and a set of linear constraints. It is important to note here, that while constraint (4) is obvious to us, if we did not specify it then spending negative amounts of money could be part of a valid solution. It is important to think about the valid values that your variables can take on for a given problem and include all necessary constraints to keep them within the required range. Given this LP formulation, we can now give it to an LP solving program and that program will give us values for x_1, x_2, x_3, x_4, which tells us how much to spend on advertising for each issue while minimizing the overall cost.

$$
\begin{array}{llllll}
\textbf{minimize} & x_1 & +x_2 & +x_3 & +x_4 & \\
\textbf{subject to} & & & & & \\
& -2,000x_1 & +8,000x_2 & +0x_3 & +10,000x_4 & \geq 50,000 \\
& 5,000x_1 & +2,000x_2 & +0x_3 & +0x_4 & \geq 100,000 \\
& 3,000x_1 & -5,000x_2 & +10,000x_3 & -2,000x_4 & \geq 25,000 \\
& x_1, x_2, x_3, x_4 & & & & \geq 0
\end{array}
$$

The next section discusses the formulation of an LP in more detail. A solution to this LP will result in the minimum amount of money that the candidate must spend on each issue. See the last section for how to use LP solvers to solve this problem.

Formulation and Solution Techniques for LPs

As we saw in the previous section, an LP consists of a linear function that we want to minimize or maximize along with a set of linear inequalities, or constraints. Note, these must be linear, so if we have a variable x_1, for example, we cannot have x_1^2 in any of the equations.

In general, we can formulate an LP as follows, where c, A, b are matrices of known coefficients and x is a matrix of values that must be determined.

$$\begin{aligned} &\textbf{max } c^T x \\ &\textbf{subject to} \\ &\quad Ax \leq b \end{aligned}$$

If we have two variables that we must solve for along with three constraints, this may look something like the following.

$$\begin{aligned} &\textbf{max } \begin{bmatrix} c_1 & c_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\ &\textbf{subject to} \\ &\quad \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \end{aligned}$$

Doing matrix multiplication on the above results in formulas similar to those seen in the example in the previous section.

$$\begin{aligned} &\textbf{max } c_1 x_1 + c_2 x_2 \\ &\textbf{subject to} \\ &\quad a_{11}x_1 + a_{12}x_2 \leq b_1 \\ &\quad a_{21}x_1 + a_{22}x_2 \leq b_2 \\ &\quad a_{31}x_1 + a_{32}x_2 \leq b_3 \end{aligned}$$

Let us represent the LP in the previous section using the matrix notation introduced here.

$$\textbf{min} \begin{bmatrix}1 & 1 & 1 & 1\end{bmatrix} \begin{bmatrix}x_1 \\ x_2 \\ x_3 \\ x_4\end{bmatrix}$$

subject to

$$\begin{bmatrix}-2,000 & 8,000 & 0 & 10,000 \\ 5,000 & 2,000 & 0 & 0 \\ 3,000 & -5,000 & 10,000 & -2,000 \\ 1 & 1 & 1 & 1\end{bmatrix} \begin{bmatrix}x_1 \\ x_2 \\ x_3 \\ x_4\end{bmatrix} \geq \begin{bmatrix}50,000 \\ 100,000 \\ 25,000 \\ 0\end{bmatrix}$$

When we work with LPs and ILPs we typically write them using the form in the previous section instead of matrix form. Other important topics for LP formulation include LP forms (standard and slack) and duality. These topics are not covered here.

Solving LPs

There are many LP solvers available, both free and commercial, that will give solutions to LPs for you. This section will just give you a basic idea about how LPs are solved in general. Solving LPs efficiently (especially LPs with many variables/constraints) is very difficult, so using LP solvers is much easier than attempting to write your own code.

Given a problem formulated as an LP, how do we go about finding the values of the vector x that minimizes or maximizes the objective function? We can think of the problem graphically. Let us consider a simple example with only two variables. Note, in this case the c vector consists of 1's.

$$\begin{aligned} \textbf{max } & x_1 + x_2 \\ \textbf{subject to } & \\ & 4x_1 + 2x_2 \leq 12 \\ & -x_1 + x_2 \leq 1 \\ & x_1 + 2x_2 \leq 4 \\ & x_{,}x_2 \geq 0 \end{aligned} \tag{5}$$

Each variable represents a dimension in a graph. In this case, we have two variables so we will have a 2D graph. Each constraint essentially splits the region. The intersection of all of the constraints is called the *feasible region*. We can see an example of the above LP in Fig. 1.

Each axis represents a variable. The variables themselves are constrained to be above 0, so we only consider positive values as shown in the graph. Each of the other lines of the graph represent a constraint, as indicated in the graph. The shaded area in the graph is the *feasible region*. Notice that this region is convex. The feasible region will always be convex. Every point in the region is a solution to the LP, meaning that it satisfies all of the constraints. We are interested in finding the optimal solution according to the objective

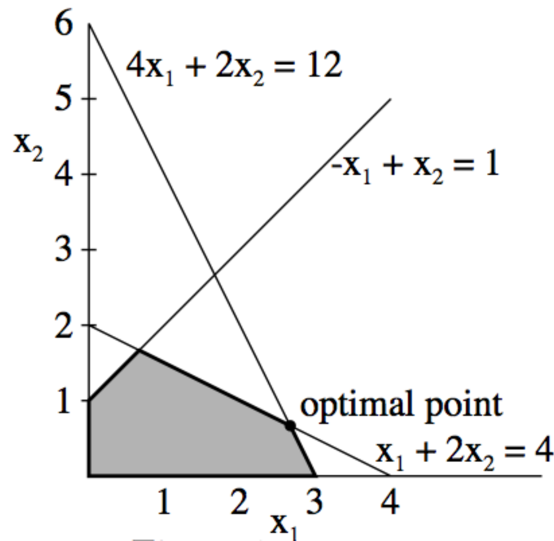

Figure 1: Example of the graph created by LP (5). Borrowed from [2].

function. One simple approach is to evaluate every point in the feasible region to find the one that gives the optimal solution. This is obviously not practical. Because of the linear constraints and convexity of the feasible region, the optimal solution will be a vertex in the graph (or adjacent vertices) where constraints intersect on the border of the feasible region, meaning we can focus our search on those vertices only. Another important property is that a local or minima or maxima is also the global minima or maxima. This means that if we find a vertex whose neighbors have higher (lower) values for the objective function, then this vertex is the global minima (maxima).

Given these facts, one of the most common techniques for solving LPs is the simplex algorithm. It basically searches the vertices of the feasible region by moving to a vertex with a higher or lower objective function value depending on the goal of the LP (maximization or minimization). When no more adjacent vertices are found that can improve the objective function, then this is the optimal solution. The simplex algorithm uses pivot rules to choose which adjacent vertex to move to if there are multiple vertices to choose from. The simplex algorithm has an exponential worst case runtime, but is typically very good in practice. There are other techniques used to solve LPs, but simplex is the most common. For example, another technique, called the interior-point method, solves the problem by traversing the interior of the feasible region instead of the vertices. There are many resources on how these algorithms work available on the web and in textbooks.

The example presented above only has 2 variables and 3 constraints. The resulting graph seems trivial to solve, but not all LPs are this simple. In general, an LP may have hundreds or thousands of variables and constraints. The resulting geometric feasible region is no longer a simple convex hull, but has many dimensions and many vertices. The number of vertices grows *exponentially* with the number of variables and constraints. Because of this, algorithms to solve LPs are very complicated in practice so as to avoid having to search an exponential number of vertices.

Using *GNU Linear Programming Kit* to Solve LPs

Now that we have seen how to formulate LPs we will use an open source program to actually solve an LP. We'll use the *GNU Linear Programming Kit* (GLPK) to solve our LPs. You can download the website and install it on any Linux system. Another popular open source program is *LP Solve*. The main purpose of this section is to introduce MathProg. The GLPK is easier to set up with MathProg than LP Solve, so we will use GLPK here.

We'll start off by solving the example problem we discussed earlier about the political campaign and then go into a network related example.

Example 1: Political Campaign

We'll start of with a very simple MathProg script to solve the LP introduced in the first section, then we'll modify it to make it more general and to introduce new concepts. The MathProg script is shown below in Listing 1.

Listing 1: Campaign LP 1 in MathProg

```
1  var x1 >= 0;
2  var x2 >= 0;
3  var x3 >= 0;
4  var x4 >= 0;
5
6  minimize advertisingCost: x1+x2+x3+x4;
7
8  s.t. urbanVotes: -2000*x1 + 8000*x2 + 10000*x4 >= 50000;
9  s.t. suburbanVotes: 5000*x1 + 2000*x2 >= 100000;
10 s.t. ruralVotes: 3000*x1 - 5000*x2 + 10000*x3 - 2000*x4 >= 25000;
11
12 solve;
13
14 printf "\n\nSolution: $%.2f\n", 1000*(x1+x2+x3+x4);
15 printf "Variables:\n";
16 printf "Roads: $%.2f\n", 1000*x1;
17 printf "Gun Control: $%.2f\n", 1000*x2;
18 printf "Farm Subisidies: $%.2f\n", 1000*x3;
19 printf "Gas Tax: $%.2f\n",  1000*x4;
20
21 end;
```

The MathProg script is almost exactly like the LP we described earlier, reproduced below. In MathProg, the *var* statement is used to declare the variables. This is what the LP is going to solve for. In this case, we have four variables given the same names as our LP formulation. Notice, we can specify the valid range of values in the variable declaration instead of using a separate constraint like we do in the LP. Next, we have the objective function statement, which in general is <*minimize*|*maximize*> <*name*>*:* <*function*>. In this case we specify we want to minimize. We give the objective function a name, *advertisingCost*, and then we specify the function itself.

$$
\begin{array}{llllll}
\textbf{minimize} & x_1 & +x_2 & +x_3 & +x_4 & \\
\textbf{subject to} & & & & & \\
 & -2,000x_1 & +8,000x_2 & +0x_3 & +10,000x_4 & \geq 50,000 \\
 & 5,000x_1 & +2,000x_2 & +0x_3 & +0x_4 & \geq 100,000 \\
 & 3,000x_1 & -5,000x_2 & +10,000x_3 & -2,000x_4 & \geq 25,000 \\
 & x_1, x_2, x_3, x_4 & & & & \geq 0
\end{array}
$$

Next come the constraints. In our LP formulation, we have three constraints that correspond to the required number of votes for each area. In MathProg, we specify a constraint with *s.t.* <*name*>: <*function*>. The *s.t.* stands for "subject to". It is easy to see how the constraints in MathProg match the constraints in our LP. With the LP written in MathProg, the next step is to call the *solve* command. Technically, this is all that we need, but it is often useful to print out any variables you are interested in, otherwise you have to parse the output file that the solver produces. The *printf* statements do exactly that. They should be self-explanatory.

After you install GLPK, the actual solver, *glpsol*, is added to your path. We can use GLPK to solve this by issuing the following command:

```
$> glpsol -m campaign.mod -o output.sol
```

The -m switch means that you are specifying a model file (more on this later), and the -o switch means you want to write the output to the specified file. When you run this command, you will see some output that provides information about what the solver is doing and when it finds the solution you will see your print statements output to the console at the end of the output. From this we can see that we should spend $18,468 on advertising for roads, $3,828 on advertising for gun control, $0 on advertising for farm subsidies, and $5,6305 on advertising for a gasoline tax. We'll just assume that it is possible to spend fractions $1,000 on advertising. There is more detailed output in the solution file (output.sol), but I will not discuss that here (see [3] if you're interested).

In this example, we basically hardcoded all the data into the LP itself. In general, however, it is better to separate the parameters to the LP (in this case the data in Table I) from the MathProg model itself. We will do that next, the code is shown in Listing 2. The first thing to notice is that there is a separate section at the end called *data*, where we store the information from Table I. You'll also notice that the constraints use the *sum* expression instead of listing the constraints individually. Lastly, we introduce two new statements, *set* and *param*. Instead of adding a data section to the file, it is possible to create a separate data file and have *glpsol* read both in with -m (for model) and -d (for data) command line switches.

First, lets discuss the *set* statement. We define two sets, one called AREAS and one called ISSUES. Here, we have two sets of strings. MathProg allows you to iterate over items in a set. We'll see how these sets are useful soon. Next, we have the *param* statement. Parameters are used to store data that is passed into the LP (as opposed to data the solver

will solve for). In this case, the data we need to store is the data shown in Table I. Line 3 only declares a parameter named VOTES, but does not populate it with data yet. The brackets on line 3 signify that this parameter is a 2-dimensional parameter (think of it like a 2D array), and to index each dimension we use elements from the AREAS and ISSUES set. So one element in this parameter is *VOTES['urban', 'guns']*, which corresponds to the cell with value 8,000 in Table I. The size of the 2D array depends on the size of the sets we define to index into it. ISSUES has 4 elements and AREAS has 3, so this is basically a 4 by 3 2D array (I will use parameter and array interchangeably, it is easier to just think of it as an array conceptually). If we wanted a 1-dimensional parameter array, then we could do something like *param TEST {ISSUES};*. This can store 4 items and to retrieve each item we use elements from ISSUES, e.g. *TEST['roads']*. We can also have simple scalar values: *param TEST2;*. TEST2 would only be able to store a single value.

Listing 2: Campaign LP 2 in MathProg

```
1  set AREAS := {'urban', 'suburban', 'rural'};
2  set ISSUES := {'roads', 'guns', 'farms', 'tax'};
3  param VOTES {ISSUES, AREAS};
4
5  var x {ISSUES} >= 0;
6
7  minimize advertisingCost: sum {i in ISSUES} x[i];
8
9  s.t. urbanVotes: sum {i in ISSUES} VOTES[i,'urban']*x[i] >= 50000;
10
11 s.t. suburbanVotes: sum {i in ISSUES} VOTES[i,'suburban']*x[i] >= 100000;
12
13 s.t. ruralVotes: sum {i in ISSUES} VOTES[i,'rural']*x[i] >= 25000;
14
15 solve;
16
17 printf "\n\nSolution: $%.2f\n", 1000*(sum {i in ISSUES} x[i]);
18 printf "Variables:\n";
19 for{i in ISSUES} {
20    printf "%s: $%.2f\n", i, 1000*x[i];
21 }
22
23 data;
24
25 param VOTES: urban suburban rural :=
26 roads          -2000 5000     3000
27 guns           8000  2000     -5000
28 farms          0     0        10000
29 tax            10000 0        -2000;
30
31 end;
```

Whenever you have data that is used by the LP and this data could change (e.g. the data in Table I could change after additional campaign research), you should store it in parameters, which you can think of as arrays. You can use sets to index into the parameters and variables (as we will see) to make writing the actual constraints easier.

In Line 5, we see a variable declaration again, but now instead of making four variables, x1, x2, x3, and x4 like last time, we define an array of variables. This array is indexed using

the set ISSUES, e.g. *x['farms']*.

In Line 7 we have our objective function just like before, but now we use the *sum* expression. Inside the brackets, we specify what the sum should iterate over, in this case it will iterate over the ISSUES set, storing each element in ISSUES in a temporary variable i. We can then use that temporary variable in the sum expression. This expression essentially expands to: $x['roads'] + x['guns'] + x['farms'] + x['tax']$. This is equivalent to $\sum_{i \in ISSUES} d_i$ in mathematical notation.

Now lets look at the constraints. The constraints are the same as in the previous MathProg script, but now we use the data stored in VOTES and use the sum expression. Lets expand line 9. This results in $VOTES['roads','urban'] * x['roads'] + VOTES['guns','urban'] * x['guns'] + VOTES['farms','urban'] * x['farms'] + VOTES['tax','urban'] * x['tax']$. If we look up those values in the VOTES table, this corresponds to the same constraint for our previous MathProg script. Again, in mathematical notation: $\sum_{i \in ISSUES} VOTES_{i,'urban'} * d_i$.

The print statements should be self-explanatory. Here we use the sum expression along with iterating over the ISSUES set instead of printing each variable one by one.

Lastly, we populate the VOTES parameter. We do this in a separate data section (which, again, can actually be stored in a separate file). VOTES is a 2D array whose indices are the ISSUES and AREAS sets. When we populate it, we have to use the entries from the indices. This part should be self explanatory (see Table I as well).

For a more detailed analysis of the output file and GLPK itself see [3]. GLPK also comes with documentation describing MathProg in detail.

Example 2: Shortest Path Weights

In this example, we'll look at a network-oriented problem. We'll use linear programming to find the shortest path distance between two nodes in a network (just the distance, not the path itself). In this problem, we are given a graph, $G = (V, E)$, a weight function $w : E \to \mathbb{R}^+$, and are asked to find the shortest path distance from a node s to a node d. To understand how this LP works, we must briefly discuss how the Bellman-Ford algorithm works. The Bellman-Ford algorithm is used to solve the single-source shortest paths problem. This algorithm repeats a "relaxation" operation on each edge for every iteration. Let d_x be the current known distance required to reach node x from the source node. Also, let $c(u, v)$ be the edge weight of edge (u, v). The relaxation step does the following on each iteration for every edge (u, v):

if $d_v > d_u + c(u, v)$ **then**
  $d_v = d_u + c(u, v)$
end if

What this means is that if our current distance estimate to node v is higher than going to node u then to node v, then our current estimate is wrong and there is a shorter path, namely the path to node u then v. This can be illustrated by Figure 2. From our source node we have some path to node v, costing d_v. We also know how to get to some node u, at cost d_u. If it is cheaper to go to u first then directly to v, we use that path instead of our current path to v.

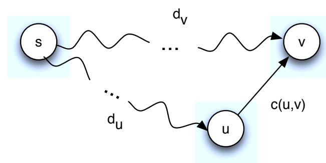

Figure 2: Illustration of relaxation step in Bellman-Ford.

The algorithm works by running this step multiple times, each step it updates its cost estimates, until it knows that its estimates are the actual shortest path distances. It can be shown that if the relaxation step is executed $|V| - 1$ times for all edges (u, v) then all of the d_i values will be the shortest path distances from a source node s. For a more detailed description, see any algorithms textbook.

Given the above relaxation, when Bellman-Ford terminates, we know that for each vertex v, $d_v \le d_u + c(u, v)$ for all edges (u, v). We can use this as our constraints for the LP. Let s be the source node and *dest* be the destination node.

$$\begin{aligned}
&\textbf{maximize } d_{dest} \\
&\textbf{subject to} \\
&\quad d_v \le d_u + c(u, v) \;\; \forall (u, v) \in E \\
&\quad d_s = 0
\end{aligned}$$

Notice that the objective function is to *maximize*, even though we are finding the *shortest* paths. Also notice that there are more than two constraints. There is actually a constraint for each edge in the network, but we're just using the "for all" notation to represent all of them. The best way to understand how this works is to write out all the constraints for a simple network. For the network in Figure 3, assume s is 1 and *dest* is 2. Write out all of the constraints and plug in the values of the cost function. You can find the shortest path by looking at the figure and you should convince yourself that the LP works. It is also trivial to modify the LP so that it finds the shortest path distance to all nodes instead of just to *dest*.

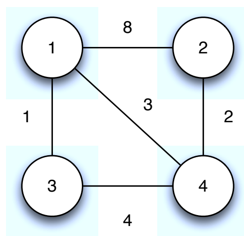

Figure 3: Network used in Listing 3.

Now lets look at the MathProg script for this problem shown below. You should understand most of the script based on the previous examples. We represent the network shown in Figure 3 by using an adjacency matrix, parameter C.

If a link does not exist in the network, then it gets a weight of 99. The second constraint in the script introduces some new notation. First, notice that immediately after the constraint name we have $\{x \text{ } in \text{ } NODES, \text{ } y \text{ } in \text{ } NODES\}$. This essentially mimics the "for all" statement in our LP. In this case, it will turn that single constraint into 16 constraints, one for each pairing of the nodes (which represent possible edges). Not all pairings of nodes are edges though (e.g. 2→3 or 3→2), so we need to make sure that the pairing is a valid edge before applying the constraint. This can be done with the if statement. It will check to make sure the cost is less than 99, meaning there does exist an edge between the two nodes.

Listing 3: Shortest Paths LP in MathProg

```
1  set NODES := {1..4};
2  var d {NODES};
3  param source := 1;
4  param dest := 2;
5
6  param C {NODES,NODES};
7
8  maximize dWeight: d[dest];
9
10 s.t. sourceNode: d[source] = 0;
11 s.t. triangleIneqality{x in NODES, y in NODES}:
12      if (C[x,y] < 99) then d[x] <= d[y] + C[y,x];
13
14 solve;
15
16 printf "\n\nShortest path from %d to %d is %f\n", source, dest, d[dest];
17
18 data;
19
20 param C: 1 2 3 4 :=
21 1 0 8 1 3
22 2 8 0 99 2
23 3 1 99 0 4
24 4 3 2 4 0;
25
26 end;
```

We'll explore an alternate (and probably more intuitive) formulation of this problem when we discuss integer linear programs.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. McGraw-Hill, 2002.

[2] T. S. Ferguson, "Linear programming: A concise introduction." [Online]. Available: http://www.usna.edu/Users/weapsys/avramov/Compressed%20sensing%20tutorial/LP.pdf

[3] R. Ceron, "The GNU linear programming kit, part 1: Introduction to linear optimization." [Online]. Available: http://www.ibm.com/developerworks/linux/library/l-glpk1/